3.2 Example Problems

The problem-solving approach has been applied to a vast array of task environments. We list some of the best known here, distinguishing between *standardized* and *real-world* problems. A **standardized problem** is intended to illustrate or exercise various problem-solving methods. It can be given a concise, exact description and hence is suitable as a benchmark for researchers to compare the performance of algorithms. A **real-world problem**, such as robot navigation, is one whose solutions people actually use, and whose formulation is idiosyncratic, not standardized, because, for example, each robot has different sensors that produce different data.

Standardized problem

Real-world problem

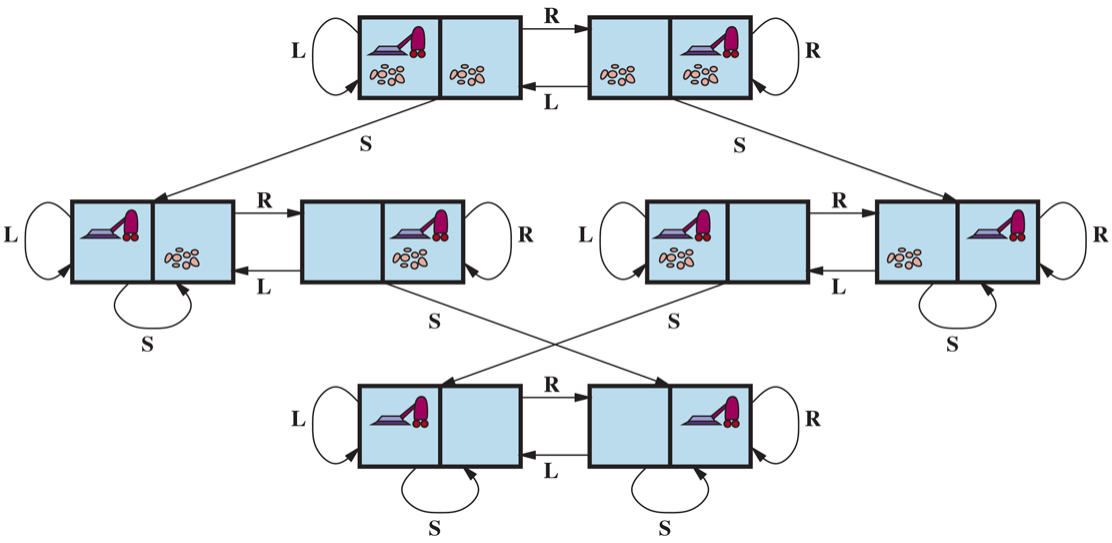
3.2.1 Standardized problems

Grid world

A **grid world** problem is a two-dimensional rectangular array of square cells in which agents can move from cell to cell. Typically the agent can move to any obstacle-free adjacent cell—horizontally or vertically and in some problems diagonally. Cells can contain objects, which the agent can pick up, push, or otherwise act upon; a wall or other impassible obstacle in a cell prevents an agent from moving into that cell. The **vacuum world** from Section 2.1 can be formulated as a grid world problem as follows:

- **STATES:** A state of the world says which objects are in which cells. For the vacuum world, the objects are the agent and any dirt. In the simple two-cell version, the agent can be in either of the two cells, and each call can either contain dirt or not, so there are $2 \cdot 2 \cdot 2 = 8$ states (see Figure 3.2). In general, a vacuum environment with n cells has $n \cdot 2^n$ states.

Figure 3.2



The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = *Left*, R = *Right*, S = *Suck*.

- **INITIAL STATE:** Any state can be designated as the initial state.
- **ACTIONS:** In the two-cell world we defined three actions: *Suck*, move *Left*, and move *Right*. In a two-dimensional multi-cell world we need more movement actions. We could add *Upward* and *Downward*, giving us four **absolute** movement actions, or we could switch to **egocentric actions**, defined relative to the viewpoint of the agent—for example, *Forward*, *Backward*, *TurnRight*, and *TurnLeft*.
- **TRANSITION MODEL:** *Suck* removes any dirt from the agent's cell; *Forward* moves the agent ahead one cell in the direction it is facing, unless it hits a wall, in which case the action has no effect. *Backward* moves the agent in the opposite direction, while *TurnRight* and *TurnLeft* change the direction it is facing by 90°.
- **GOAL STATES:** The states in which every cell is clean.
- **ACTION COST:** Each action costs 1.

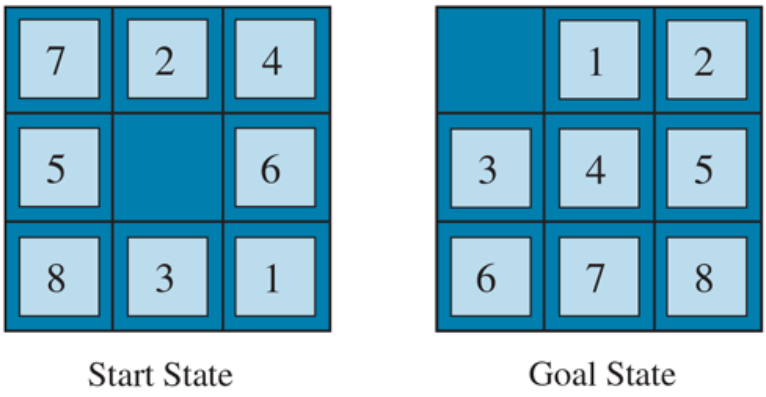
Sokoban puzzle

Another type of grid world is the **sokoban puzzle**, in which the agent's goal is to push a number of boxes, scattered about the grid, to designated storage locations. There can be at most one box per cell. When an agent moves forward into a cell containing a box and there is an empty cell on the other side of the box, then both the box and the agent move forward. The agent can't push a box into another box or a wall. For a world with n non-obstacle cells and b boxes, there are $n \times n!/(b!(n-b)!)$ states; for example on an 8×8 grid with a dozen boxes, there are over 200 trillion states.

In a **sliding-tile puzzle**, a number of tiles (sometimes called blocks or pieces) are arranged in a grid with one or more blank spaces so that some of the tiles can slide into the blank space. One variant is the Rush Hour puzzle, in which cars and trucks slide around a 6×6 grid in an attempt to free a car from the traffic jam. Perhaps the best-known variant is the **8-puzzle** (see Figure 3.3), which consists of a 3×3 grid with eight numbered tiles and one blank space, and the **15-puzzle** on a 4×4 grid. The object is to reach a specified goal state, such as the one shown on the right of the figure. The standard formulation of the 8 puzzle is as follows:

- **STATES:** A state description specifies the location of each of the tiles.
- **INITIAL STATE:** Any state can be designated as the initial state. Note that a parity property partitions the state space—any given goal can be reached from exactly half of the possible initial states (see Exercise 3.PART).
- **ACTIONS:** While in the physical world it is a tile that slides, the simplest way of describing an action is to think of the blank space moving *Left*, *Right*, *Up*, or *Down*. If the blank is at an edge or corner then not all actions will be applicable.
- **TRANSITION MODEL:** Maps a state and action to a resulting state; for example, if we apply *Left* to the start state in Figure 3.3, the resulting state has the 5 and the blank switched.

Figure 3.3



A typical instance of the 8-puzzle.

- **GOAL STATE:** Although any state could be the goal, we typically specify a state with the numbers in order, as in Figure 3.3.
- **ACTION COST:** Each action costs 1.

Sliding-tile puzzle

8-puzzle

15-puzzle

Note that every problem formulation involves abstractions. The 8-puzzle actions are abstracted to their beginning and final states, ignoring the intermediate locations where the tile is sliding. We have abstracted away actions such as shaking the board when tiles get stuck and ruled out extracting the tiles with a knife and putting them back again. We are left with a description of the rules, avoiding all the details of physical manipulations.

Our final standardized problem was devised by Donald Knuth (1964) and illustrates how infinite state spaces can arise. Knuth conjectured that starting with the number 4, a

sequence of square root, floor, and factorial operations can reach any desired positive integer. For example, we can reach 5 from 4 as follows:

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5.$$

The problem definition is simple:

- **STATES:** Positive real numbers.
- **INITIAL STATE:** 4.
- **ACTIONS:** Apply square root, floor, or factorial operation (factorial for integers only).
- **TRANSITION MODEL:** As given by the mathematical definitions of the operations.
- **GOAL STATE:** The desired positive integer.
- **ACTION COST:** Each action costs 1.

The state space for this problem is infinite: for any integer greater than 2 the factorial operator will always yield a larger integer. The problem is interesting because it explores very large numbers: the shortest path to 5 goes through $(4!)! = 620{,}448{,}401{,}733{,}239{,}439{,}360{,}000$. Infinite state spaces arise frequently in tasks involving the generation of mathematical expressions, circuits, proofs, programs, and other recursively defined objects.

3.2.2 Real-world problems

We have already seen how the **route-finding problem** is defined in terms of specified locations and transitions along edges between them. Route-finding algorithms are used in a variety of applications. Some, such as Web sites and in-car systems that provide driving directions, are relatively straightforward extensions of the Romania example. (The main complications are varying costs due to traffic-dependent delays, and rerouting due to road closures.) Others, such as routing video streams in computer networks, military operations planning, and airline travel-planning systems, involve much more complex specifications. Consider the airline travel problems that must be solved by a travel-planning Web site:

- **STATES:** Each state obviously includes a location (e.g., an airport) and the current time. Furthermore, because the cost of an action (a flight segment) may depend on previous

segments, their fare bases, and their status as domestic or international, the state must record extra information about these "historical" aspects.

- **INITIAL STATE:** The user's home airport.
- **ACTIONS:** Take any flight from the current location, in any seat class, leaving after the current time, leaving enough time for within-airport transfer if needed.
- **TRANSITION MODEL:** The state resulting from taking a flight will have the flight's destination as the new location and the flight's arrival time as the new time.
- **GOAL STATE:** A destination city. Sometimes the goal can be more complex, such as "arrive at the destination on a nonstop flight."
- **ACTION COST:** A combination of monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer reward points, and so on.

Commercial travel advice systems use a problem formulation of this kind, with many additional complications to handle the airlines' byzantine fare structures. Any seasoned traveler knows, however, that not all air travel goes according to plan. A really good system should include contingency plans—what happens if this flight is delayed and the connection is missed?

Touring problems describe a set of locations that must be visited, rather than a single goal destination. The **traveling salesperson problem (TSP)** is a touring problem in which every city on a map must be visited. The aim is to find a tour with cost $< C$ (or in the optimization version, to find a tour with the lowest cost possible). An enormous amount of effort has been expended to improve the capabilities of TSP algorithms. The algorithms can also be extended to handle fleets of vehicles. For example, a search and optimization algorithm for routing school buses in Boston saved $5 million, cut traffic and air pollution, and saved time for drivers and students (Bertsimas *et al.*, 2019). In addition to planning trips, search algorithms have been used for tasks such as planning the movements of automatic circuit-board drills and of stocking machines on shop floors.

Touring problem

Traveling salesperson problem (TSP)

A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield. The layout problem comes after the logical design phase and is usually split into two parts: **cell layout** and **channel routing**. In cell layout, the primitive components of the circuit are grouped into cells, each of which performs some recognized function. Each cell has a fixed footprint (size and shape) and requires a certain number of connections to each of the other cells. The aim is to place the cells on the chip so that they do not overlap and so that there is room for the connecting wires to be placed between the cells. Channel routing finds a specific route for each wire through the gaps between the cells. These search problems are extremely complex, but definitely worth solving.

VLSI layout

Robot navigation is a generalization of the route-finding problem described earlier. Rather than following distinct paths (such as the roads in Romania), a robot can roam around, in effect making its own paths. For a circular robot moving on a flat surface, the space is essentially two-dimensional. When the robot has arms and legs that must also be controlled, the search space becomes many-dimensional—one dimension for each joint angle. Advanced techniques are required just to make the essentially continuous search space finite (see Chapter 26). In addition to the complexity of the problem, real robots must also deal with errors in their sensor readings and motor controls, with partial observability, and with other agents that might alter the environment.

Robot navigation

Automatic assembly sequencing of complex objects (such as electric motors) by a robot has been standard industry practice since the 1970s. Algorithms first find a feasible assembly sequence and then work to optimize the process. Minimizing the amount of manual human labor on the assembly line can produce significant savings in time and cost. In assembly problems, the aim is to find an order in which to assemble the parts of some object. If the wrong order is chosen, there will be no way to add some part later in the sequence without undoing some of the work already done. Checking an action in the sequence for feasibility is a difficult geometrical search problem closely related to robot navigation. Thus, the generation of legal actions is the expensive part of assembly sequencing. Any practical algorithm must avoid exploring all but a tiny fraction of the state space. One important assembly problem is **protein design**, in which the goal is to find a sequence of amino acids that will fold into a three-dimensional protein with the right properties to cure some disease.

Automatic assembly sequencing

Protein design

3.3 Search Algorithms

A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure. In this chapter we consider algorithms that superimpose a **search tree** over the state-space graph, forming various paths from the initial state, trying to find a path that reaches a goal state. Each **node** in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions. The root of the tree corresponds to the initial state of the problem.

Search algorithm

Node

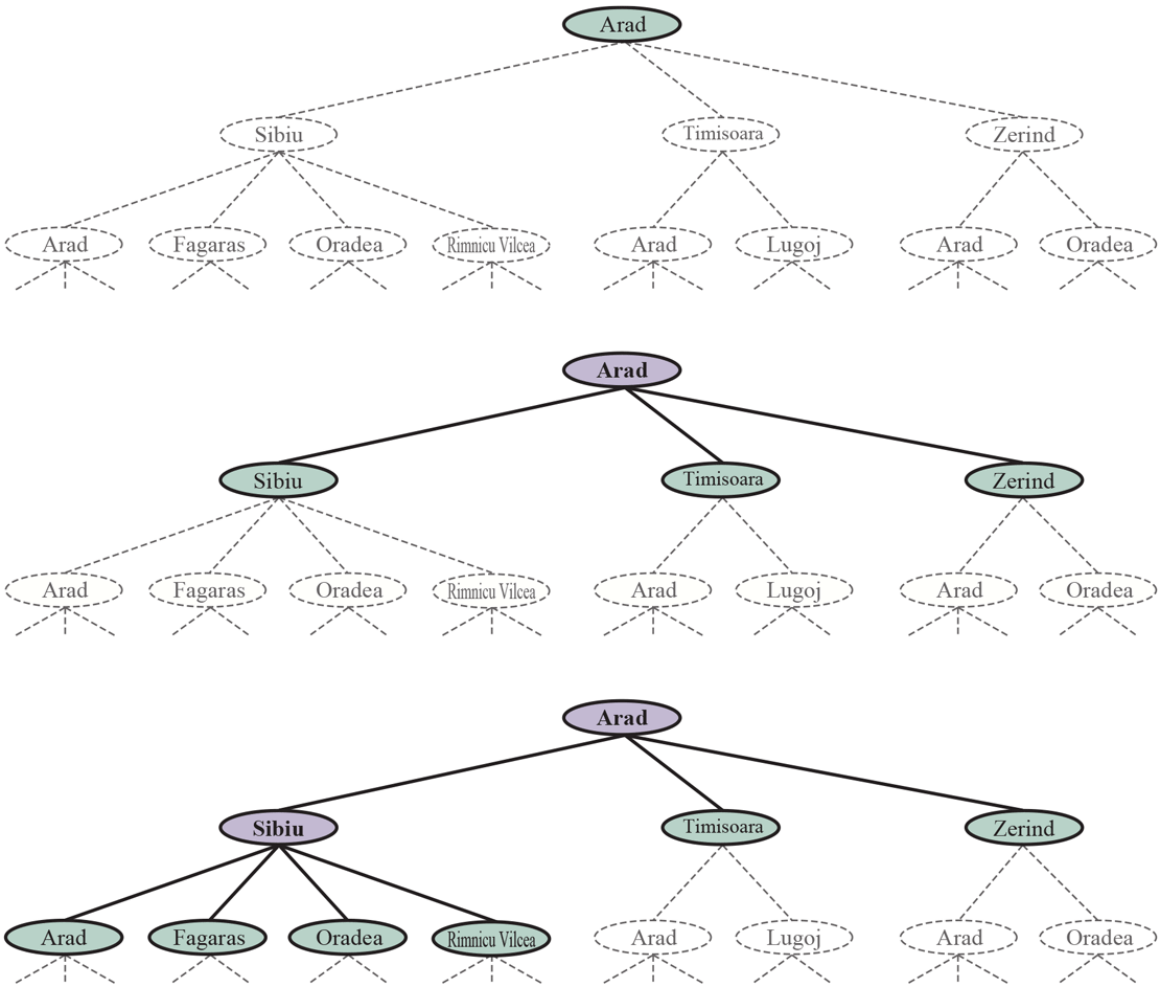
It is important to understand the distinction between the state space and the search tree. The state space describes the (possibly infinite) set of states in the world, and the actions that allow transitions from one state to another. The search tree describes paths between these states, reaching towards the goal. The search tree may have multiple paths to (and thus multiple nodes for) any given state, but each node in the tree has a unique path back to the root (as in all trees).

Expand

Figure 3.4 shows the first few steps in finding a path from Arad to Bucharest. The root node of the search tree is at the initial state, *Arad*. We can **expand** the node, by considering the available ACTIONS for that state, using the RESULT function to see where those actions lead

to, and **generating** a new node (called a **child node** or **successor node**) for each of the resulting states. Each child node has *Arad* as its **parent node**.

Figure 3.4



Three partial search trees for finding a route from Arad to Bucharest. Nodes that have been *expanded* are lavender with bold letters; nodes on the frontier that have been *generated* but not yet expanded are in green; the set of states corresponding to these two types of nodes are said to have been *reached*. Nodes that could be generated next are shown in faint dashed lines. Notice in the bottom tree there is a cycle from Arad to Sibiu to Arad; that can't be an optimal path, so search should not continue from there.

Generating

Child node

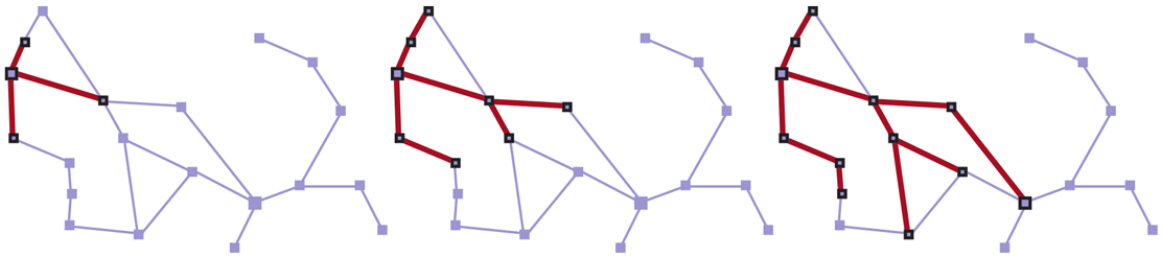
Successor node

Parent node

Now we must choose which of these three child nodes to consider next. This is the essence of search—following up one option now and putting the others aside for later. Suppose we choose to expand Sibiu first. Figure 3.4 (bottom) shows the result: a set of 6 unexpanded nodes (outlined in bold). We call this the **frontier** of the search tree. We say that any state that has had a node generated for it has been **reached** (whether or not that node has been expanded).[5] Figure 3.5 shows the search tree superimposed on the state-space graph.

5 Some authors call the frontier the **open list**, which is both geographically less evocative and computationally less appropriate, because a queue is more efficient than a list here. Those authors use the term **closed list** to refer to the set of previously expanded nodes, which in our terminology would be the *reached* nodes minus the *frontier*.

Figure 3.5



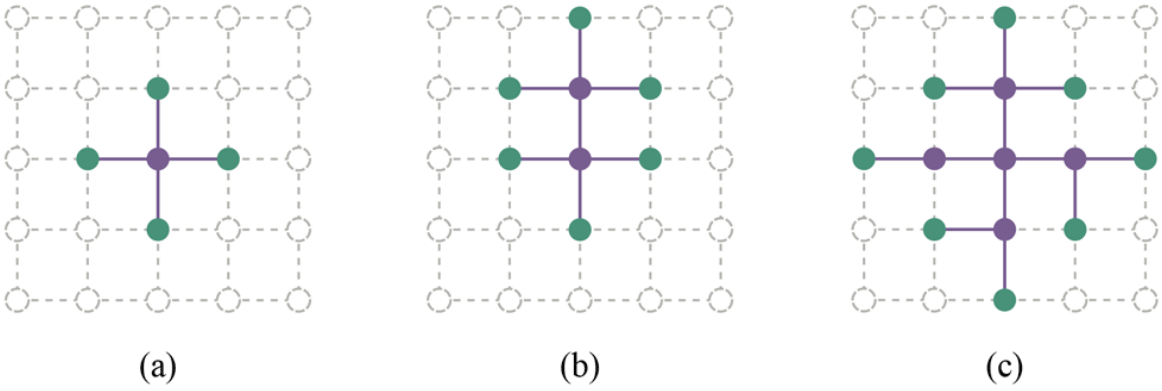
A sequence of search trees generated by a graph search on the Romania problem of Figure 3.1. At each stage, we have expanded every node on the frontier, extending every path with all applicable actions that don't result in a state that has already been reached. Notice that at the third stage, the topmost city (Oradea) has two successors, both of which have already been reached by other paths, so no paths are extended from Oradea.

Frontier

Reached

Note that the frontier **separates** two regions of the state-space graph: an interior region where every state has been expanded, and an exterior region of states that have not yet been reached. This property is illustrated in Figure 3.6.

Figure 3.6



The separation property of graph search, illustrated on a rectangular-grid problem. The frontier (green) separates the interior (lavender) from the exterior (faint dashed). The frontier is the set of nodes (and corresponding states) that have been reached but not yet expanded; the interior is the set of nodes (and corresponding states) that have been expanded; and the exterior is the set of states that have not been reached. In (a), just the root has been expanded. In (b), the top frontier node is expanded. In (c), the remaining successors of the root are expanded in clockwise order.

Separator

3.3.1 Best-first search

How do we decide which node from the frontier to expand next? A very general approach is called **best-first search**, in which we choose a node, n, with minimum value of some

evaluation function, $f(n)$. Figure 3.7 shows the algorithm. On each iteration we choose a node on the frontier with minimum $f(n)$ value, return it if its state is a goal state, and otherwise apply EXPAND to generate child nodes. Each child node is added to the frontier if it has not been reached before, or is re-added if it is now being reached with a path that has a lower path cost than any previous path. The algorithm returns either an indication of failure, or a node that represents a path to a goal. By employing different $f(n)$ functions, we get different specific algorithms, which this chapter will cover.

Figure 3.7

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

The best-first search algorithm, and the function for expanding a node. The data structures used here are described in Section 3.3.2. See Appendix B for yield.

Best-first search

Evaluation function

3.3.2 Search data structures

Search algorithms require a data structure to keep track of the search tree. A **node** in the tree is represented by a data structure with four components:

- *node*.STATE: the state to which the node corresponds;
- *node*.PARENT: the node in the tree that generated this node;
- *node*.ACTION: the action that was applied to the parent's state to generate this node;
- *node*.PATH-COST: the total cost of the path from the initial state to this node. In mathematical formulas, we use *g(node)* as a synonym for PATH-COST.

Following the PARENT pointers back from a node allows us to recover the states and actions along the path to that node. Doing this from a goal node gives us the solution.

We need a data structure to store the **frontier**. The appropriate choice is a **queue** of some kind, because the operations on a frontier are:

- IS-EMPTY(*frontier*) returns true only if there are no nodes in the frontier.
- POP(*frontier*) removes the top node from the frontier and returns it.
- TOP(*frontier*) returns (but does not remove) the top node of the frontier.
- ADD(*node, frontier*) inserts node into its proper place in the queue.

Queue

Three kinds of queues are used in search algorithms:

- A **priority queue** first pops the node with the minimum cost according to some evaluation function, f. It is used in best-first search.

Priority queue

- A **FIFO queue** or first-in-first-out queue first pops the node that was added to the queue first; we shall see it is used in breadth-first search.

FIFO queue

- A **LIFO queue** or last-in-first-out queue (also known as a **stack**) pops first the most recently added node; we shall see it is used in depth-first search.

LIFO queue

Stack

The reached states can be stored as a lookup table (e.g. a hash table) where each key is a state and each value is the node for that state.

3.3.3 Redundant paths

The search tree shown in Figure 3.4 (bottom) includes a path from Arad to Sibiu and back to Arad again. We say that *Arad* is a **repeated state** in the search tree, generated in this case by a **cycle** (also known as a **loopy path**). So even though the state space has only 20 states, the complete search tree is *infinite* because there is no limit to how often one can traverse a loop.

Repeated state

Cycle

Loopy path

A cycle is a special case of a **redundant path**. For example, we can get to Sibiu via the path Arad–Sibiu (140 miles long) or the path Arad–Zerind–Oradea–Sibiu (297 miles long). This second path is redundant—it's just a worse way to get to the same state—and need not be considered in our quest for optimal paths.

Redundant path

Consider an agent in a 10×10 grid world, with the ability to move to any of 8 adjacent squares. If there are no obstacles, the agent can reach any of the 100 squares in 9 moves or fewer. But the number of paths of length 9 is almost 8^9 (a bit less because of the edges of the grid), or more than 100 million. In other words, the average cell can be reached by over a million redundant paths of length 9, and if we eliminate redundant paths, we can complete a search roughly a million times faster. As the saying goes, *algorithms that cannot remember the past are doomed to repeat it*. There are three approaches to this issue.

First, we can remember all previously reached states (as best-first search does), allowing us to detect all redundant paths, and keep only the best path to each state. This is appropriate for state spaces where there are many redundant paths, and is the preferred choice when the table of reached states will fit in memory.

Graph search

Tree-like search

Second, we can not worry about repeating the past. There are some problem formulations where it is rare or impossible for two paths to reach the same state. An example would be an assembly problem where each action adds a part to an evolving assemblage, and there is an ordering of parts so that it is possible to add A and then B, but not B and then A. For those problems, we could save memory space if we *don't* track reached states and we don't check for redundant paths. We call a search algorithm a **graph search** if it checks for redundant paths and a **tree-like search**[6] if it does not check. The BEST-FIRST-SEARCH algorithm in Figure 3.7 is a graph search algorithm; if we remove all references to *reached* we get a tree-like search that uses less memory but will examine redundant paths to the same state, and thus will run slower.

6 We say "tree-like search" because the state space is still the same graph no matter how we search it; we are just choosing to treat it *as if* it were a tree, with only one path from each node back to the root.

Third, we can compromise and check for cycles, but not for redundant paths in general. Since each node has a chain of parent pointers, we can check for cycles with no need for additional memory by following up the chain of parents to see if the state at the end of the path has appeared earlier in the path. Some implementations follow this chain all the way up, and thus eliminate all cycles; other implementations follow only a few links (e.g., to the parent, grandparent, and great-grandparent), and thus take only a constant amount of time, while eliminating all short cycles (and relying on other mechanisms to deal with long cycles).

3.3.4 Measuring problem-solving performance

Before we get into the design of various search algorithms, we will consider the criteria used to choose among them. We can evaluate an algorithm's performance in four ways:

- **COMPLETENESS:** Is the algorithm guaranteed to find a solution when there is one, and to correctly report failure when there is not?

Completeness

- **COST OPTIMALITY:** Does it find a solution with the lowest path cost of all solutions?[7]

[7] Some authors use the term "admissibility" for the property of finding the lowest-cost solution, and some use just "optimality," but that can be confused with other types of optimality.

Cost optimality

- **TIME COMPLEXITY:** How long does it take to find a solution? This can be measured in seconds, or more abstractly by the number of states and actions considered.

Time complexity

- **SPACE COMPLEXITY:** How much memory is needed to perform the search?

Space complexity

To understand completeness, consider a search problem with a single goal. That goal could be anywhere in the state space; therefore a complete algorithm must be capable of systematically exploring every state that is reachable from the initial state. In finite state

spaces that is straightforward to achieve: as long as we keep track of paths and cut off ones that are cycles (e.g. Arad to Sibiu to Arad), eventually we will reach every reachable state.

In infinite state spaces, more care is necessary. For example, an algorithm that repeatedly applied the "factorial" operator in Knuth's "4" problem would follow an infinite path from 4 to 4! to (4!)!, and so on. Similarly, on an infinite grid with no obstacles, repeatedly moving forward in a straight line also follows an infinite path of new states. In both cases the algorithm never returns to a state it has reached before, but is incomplete because wide expanses of the state space are never reached.

To be complete, a search algorithm must be **systematic** in the way it explores an infinite state space, making sure it can eventually reach any state that is connected to the initial state. For example, on the infinite grid, one kind of systematic search is a spiral path that covers all the cells that are s steps from the origin before moving out to cells that are $s + 1$ steps away. Unfortunately, in an infinite state space with no solution, a sound algorithm needs to keep searching forever; it can't terminate because it can't know if the next state will be a goal.

Systematic

Time and space complexity are considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state-space graph, $|V| + |E|$, where $|V|$ is the number of vertices (state nodes) of the graph and $|E|$ is the number of edges (distinct state/action pairs). This is appropriate when the graph is an explicit data structure, such as the map of Romania. But in many AI problems, the graph is represented only *implicitly* by the initial state, actions, and transition model. For an implicit state space, complexity can be measured in terms of d, the **depth** or number of actions in an optimal solution; m, the maximum number of actions in any path; and b, the **branching factor** or number of successors of a node that need to be considered.

Depth

Branching factor

3.4 Uninformed Search Strategies

An uninformed search algorithm is given no clue about how close a state is to the goal(s). For example, consider our agent in Arad with the goal of reaching Bucharest. An uninformed agent with no knowledge of Romanian geography has no clue whether going to Zerind or Sibiu is a better first step. In contrast, an informed agent (Section 3.5) who knows the location of each city knows that Sibiu is much closer to Bucharest and thus more likely to be on the shortest path.

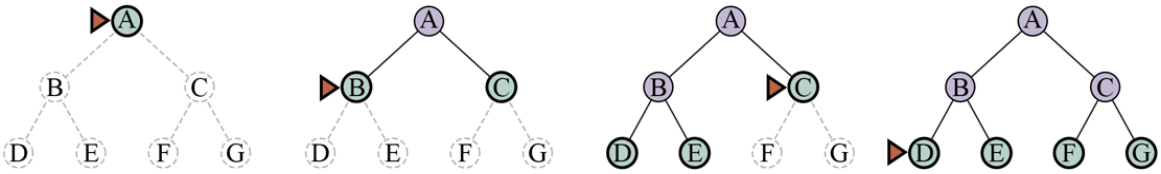
3.4.1 Breadth-first search

When all actions have the same cost, an appropriate strategy is **breadth-first search**, in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on. This is a systematic search strategy that is therefore complete even on infinite state spaces. We could implement breadth-first search as a call to BEST-FIRST-SEARCH where the evaluation function $f(n)$ is the depth of the node—that is, the number of actions it takes to reach the node.

Breadth-first search

However, we can get additional efficiency with a couple of tricks. A first-in-first-out queue will be faster than a priority queue, and will give us the correct order of nodes: new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. In addition, *reached* can be a set of states rather than a mapping from states to nodes, because once we've reached a state, we can never find a better path to the state. That also means we can do an **early goal test**, checking whether a node is a solution as soon as it is *generated*, rather than the **late goal test** that best-first search uses, waiting until a node is popped off the queue. Figure 3.8 shows the progress of a breadth-first search on a binary tree, and Figure 3.9 shows the algorithm with the early-goal efficiency enhancements.

Figure 3.8



Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

Figure 3.9

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

Breadth-first search and uniform-cost search algorithms.

Early goal test

Late goal test

Breadth-first search always finds a solution with a minimal number of actions, because when it is generating nodes at depth d, it has already generated all the nodes at depth $d-1$, so if one of them were a solution, it would have been found. That means it is cost-optimal for problems where all actions have the same cost, but not for problems that don't have that property. It is complete in either case. In terms of time and space, imagine searching a uniform tree where every state has b successors. The root of the search tree generates b nodes, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on. Now suppose that the solution is at depth d. Then the total number of nodes generated is

$$1 + b + b^2 + b^3 + \cdots + b^d = O\left(b^d\right)$$

All the nodes remain in memory, so both time and space complexity are $O(b^d)$. Exponential bounds like that are scary. As a typical real-world example, consider a problem with branching factor $b = 10$, processing speed 1 million nodes/second, and memory requirements of 1 Kbyte/node. A search to depth $d = 10$ would take less than 3 hours, but would require 10 terabytes of memory. *The memory requirements are a bigger problem for breadth-first search than the execution time*. But time is still an important factor. At depth $d = 14$, even with infinite memory, the search would take 3.5 years. In general, *exponential-complexity search problems cannot be solved by uninformed search for any but the smallest instances*.

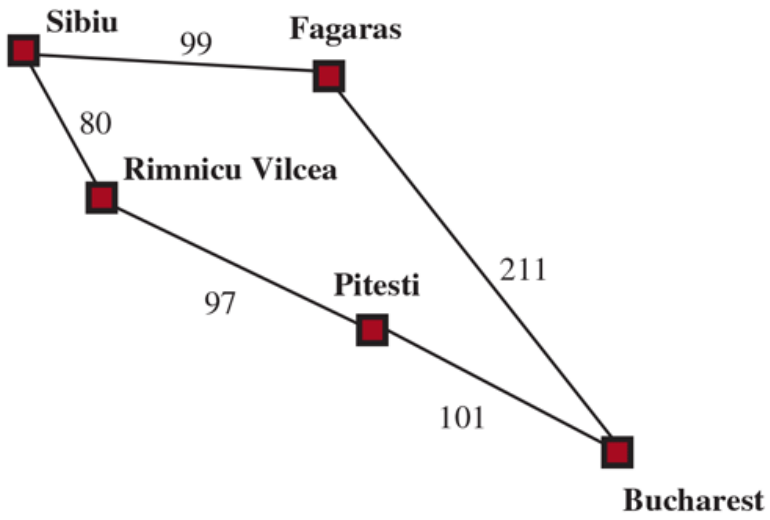
3.4.2 Dijkstra's algorithm or uniform-cost search

When actions have different costs, an obvious choice is to use best-first search where the evaluation function is the cost of the path from the root to the current node. This is called Dijkstra's algorithm by the theoretical computer science community, and **uniform-cost search** by the AI community. The idea is that while breadth-first search spreads out in waves of uniform depth—first depth 1, then depth 2, and so on—uniform-cost search spreads out in waves of uniform path-cost. The algorithm can be implemented as a call to BEST-FIRST-SEARCH with PATH-COST as the evaluation function, as shown in Figure 3.9.

Uniform-cost search

Consider Figure 3.10, where the problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost $80 + 97 = 177$. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost $99 + 211 = 310$. Bucharest is the goal, but the algorithm tests for goals only when it expands a node, not when it generates a node, so it has not yet detected that this is a path to the goal.

Figure 3.10



Part of the Romania state space, selected to illustrate uniform-cost search.

The algorithm continues on, choosing Pitesti for expansion next and adding a second path to Bucharest with cost $80 + 97 + 101 = 278$. It has a lower cost, so it replaces the previous path in *reached* and is added to the *frontier*. It turns out this node now has the lowest cost, so it is considered next, found to be a goal, and returned. Note that if we had checked for a goal upon generating a node rather than when expanding the lowest-cost node, then we would have returned a higher-cost path (the one through Fagaras).

The complexity of uniform-cost search is characterized in terms of C^*, the cost of the optimal solution,[8] and ϵ, a lower bound on the cost of each action, with $\epsilon > 0$. Then the algorithm's worst-case time and space complexity is $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, which can be much greater than b^d. This is because uniform-cost search can explore large trees of actions with low costs before exploring paths involving a high-cost and perhaps useful action. When all action costs are equal, $b^{1+\lfloor C^*/\epsilon \rfloor}$ is just b^{d+1}, and uniform-cost search is similar to breadth-first search.

8 Here, and throughout the book, the "star" in C^* means an optimal value for C.

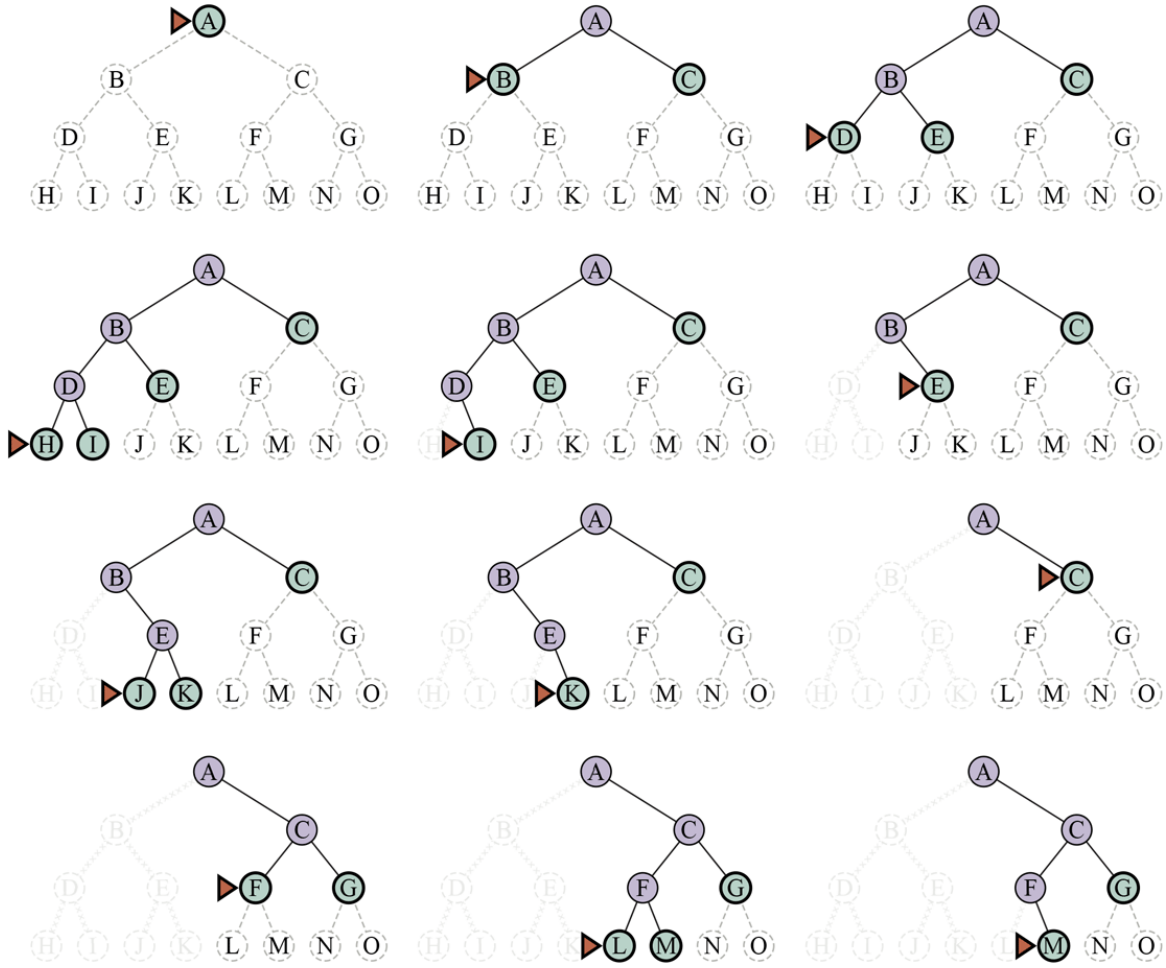
Uniform-cost search is complete and is cost-optimal, because the first solution it finds will have a cost that is at least as low as the cost of any other node in the frontier. Uniform-cost search considers all paths systematically in order of increasing cost, never getting caught going down a single infinite path (assuming that all action costs are $> \epsilon > 0$).

3.4.3 Depth-first search and the problem of memory

Depth-first search

Depth-first search always expands the *deepest* node in the frontier first. It could be implemented as a call to BEST-FIRST-SEARCH where the evaluation function f is the negative of the depth. However, it is usually implemented not as a graph search but as a tree-like search that does not keep a table of reached states. The progress of the search is illustrated in Figure 3.11; search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. The search then "backs up" to the next deepest node that still has unexpanded successors. Depth-first search is not cost-optimal; it returns the first solution it finds, even if it is not cheapest.

Figure 3.11



A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

For finite state spaces that are trees it is efficient and complete; for acyclic state spaces it may end up expanding the same state many times via different paths, but will (eventually) systematically explore the entire space.

In cyclic state spaces it can get stuck in an infinite loop; therefore some implementations of depth-first search check each new node for cycles. Finally, in infinite state spaces, depth-first search is not systematic: it can get stuck going down an infinite path, even if there are no cycles. Thus, depth-first search is incomplete.

With all this bad news, why would anyone consider using depth-first search rather than breadth-first or best-first? The answer is that for problems where a tree-like search is feasible, depth-first search has much smaller needs for memory. We don't keep a *reached*

table at all, and the frontier is very small: think of the frontier in breadth-first search as the surface of an ever-expanding sphere, while the frontier in depth-first search is just a radius of the sphere.

For a finite tree-shaped state-space like the one in Figure 3.11, a depth-first tree-like search takes time proportional to the number of states, and has memory complexity of only $O(bm)$, where b is the branching factor and m is the maximum depth of the tree. Some problems that would require exabytes of memory with breadth-first search can be handled with only kilobytes using depth-first search. Because of its parsimonious use of memory, depth-first tree-like search has been adopted as the basic workhorse of many areas of AI, including constraint satisfaction (Chapter 6), propositional satisfiability (Chapter 7), and logic programming (Chapter 9).

A variant of depth-first search called **backtracking search** uses even less memory. (See Chapter 6 for more details.) In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. In addition, successors are generated by *modifying* the current state description directly rather than allocating memory for a brand-new state. This reduces the memory requirements to just one state description and a path of $O(m)$ actions; a significant savings over $O(bm)$ states for depth-first search. With backtracking we also have the option of maintaining an efficient set data structure for the states on the current path, allowing us to check for a cyclic path in $O(1)$ time rather than $O(m)$. For backtracking to work, we must be able to *undo* each action when we backtrack. Backtracking is critical to the success of many problems with large state descriptions, such as robotic assembly.

Backtracking search

3.4.4 Depth-limited and iterative deepening search

To keep depth-first search from wandering down an infinite path, we can use **depth-limited search**, a version of depth-first search in which we supply a depth limit, ℓ, and treat all nodes at depth ℓ as if they had no successors (see Figure 3.12). The time complexity is

$O(b^\ell)$ and the space complexity is $O(b\ell)$. Unfortunately, if we make a poor choice for ℓ the algorithm will fail to reach the solution, making it incomplete again.

Figure 3.12

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

Iterative deepening and depth-limited tree-like search. Iterative deepening repeatedly applies depth-limited search with increasing limits. It returns one of three different types of values: either a solution node; or *failure*, when it has exhausted all nodes and proved there is no solution at any depth; or *cutoff*, to mean there might be a solution at a deeper depth than ℓ. This is a tree-like search algorithm that does not keep track of *reached* states, and thus uses much less memory than best-first search, but runs the risk of visiting the same state multiple times on different paths. Also, if the IS-CYCLE check does not check *all* cycles, then the algorithm may get caught in a loop.

Depth-limited search

Since depth-first search is a tree-like search, we can't keep it from wasting time on redundant paths in general, but we can eliminate cycles at the cost of some computation time. If we look only a few links up in the parent chain we can catch most cycles; longer cycles are handled by the depth limit.

Sometimes a good depth limit can be chosen based on knowledge of the problem. For example, on the map of Romania there are 20 cities. Therefore, $\ell = 19$ is a valid limit. But if we studied the map carefully, we would discover that any city can be reached from any other city in at most 9 actions. This number, known as the **diameter** of the state-space graph, gives us a better depth limit, which leads to a more efficient depth-limited search. However, for most problems we will not know a good depth limit until we have solved the problem.

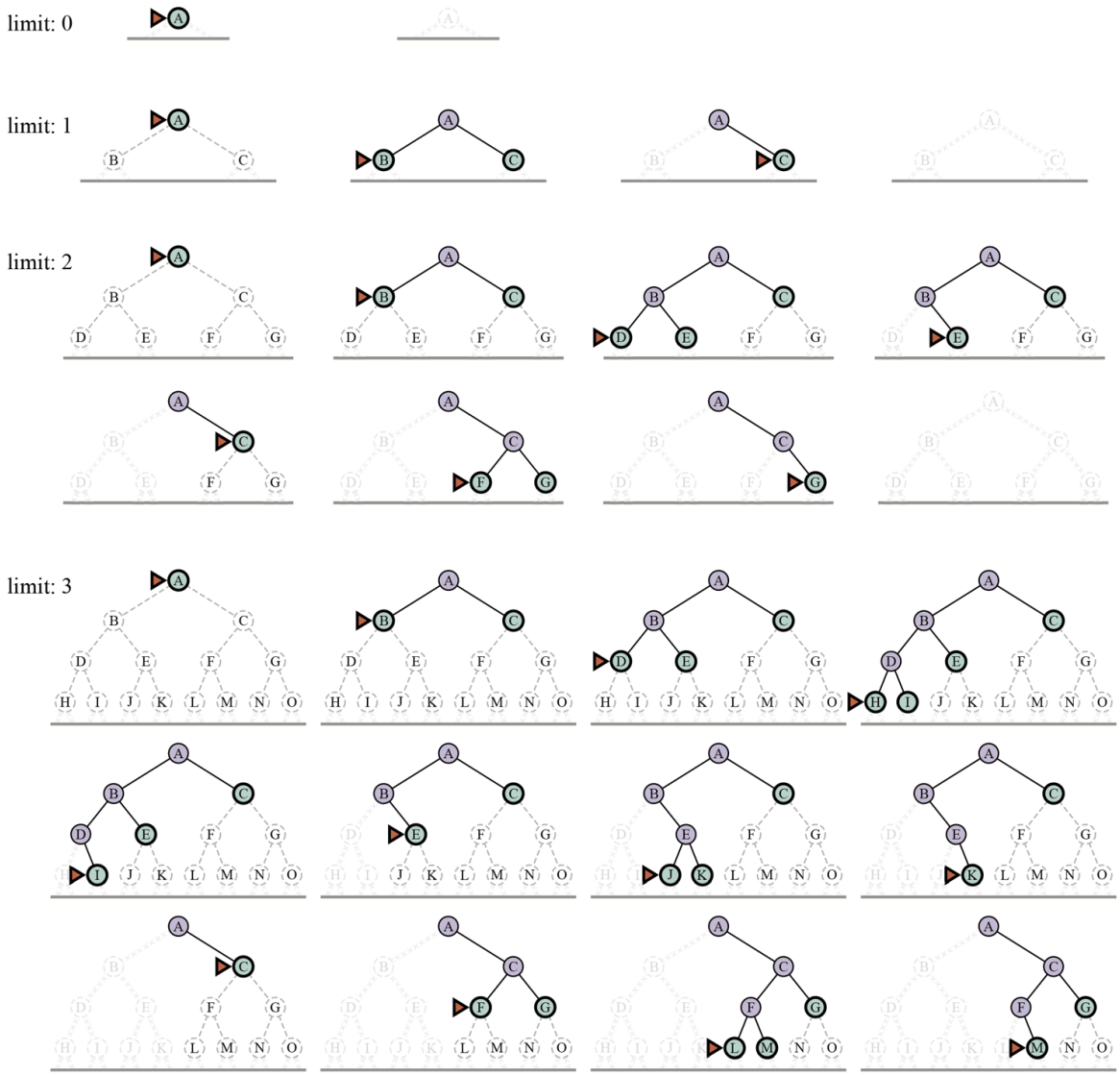
Diameter

Iterative deepening search solves the problem of picking a good value for ℓ by trying all values: first 0, then 1, then 2, and so on—until either a solution is found, or the depth-limited search returns the *failure* value rather than the *cutoff* value. The algorithm is shown in Figure 3.12. Iterative deepening combines many of the benefits of depth-first and breadth-first search. Like depth-first search, its memory requirements are modest: $O(bd)$ when there is a solution, or $O(bm)$ on finite state spaces with no solution. Like breadth-first search, iterative deepening is optimal for problems where all actions have the same cost, and is complete on finite acyclic state spaces, or on any finite state space when we check nodes for cycles all the way up the path.

Iterative deepening search

The time complexity is $O(b^d)$ when there is a solution, or $O(b^m)$ when there is none. Each iteration of iterative deepening search generates a new level, in the same way that breadth-first search does, but breadth-first does this by storing all nodes in memory, while iterative-deepening does it by repeating the previous levels, thereby saving memory at the cost of more time. Figure 3.13 shows four iterations of iterative-deepening search on a binary search tree, where the solution is found on the fourth iteration.

Figure 3.13



Four iterations of iterative deepening search for goal **M** on a binary tree, with the depth limit varying from 0 to 3. Note the interior nodes form a single path. The triangle marks the node to expand next; green nodes with dark outlines are on the frontier; the very faint nodes provably can't be part of a solution with this depth limit.

Iterative deepening search may seem wasteful because states near the top of the search tree are re-generated multiple times. But for many state spaces, most of the nodes are in the bottom level, so it does not matter much that the upper levels are repeated. In an iterative deepening search, the nodes on the bottom level (depth d) are generated once, those on the next-to-bottom level are generated twice, and so on, up to the children of the root, which are generated d times. So the total number of nodes generated in the worst case is

$$N(\text{IDS}) \quad = \quad (d)b^1 + (d-1)b^2 + (d-2)b^3 \; \ldots + b^d,$$

which gives a time complexity of $O(b^d)$—asymptotically the same as breadth-first search. For example, if $b = 10$ and $d = 5$, the numbers are

$$N(\text{IDS}) = 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}450$$
$$N(\text{BFS}) = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}110.$$

If you are really concerned about the repetition, you can use a hybrid approach that runs breadth-first search until almost all the available memory is consumed, and then runs iterative deepening from all the nodes in the frontier. *In general, iterative deepening is the preferred uninformed search method when the search state space is larger than can fit in memory and the depth of the solution is not known*.

3.4.5 Bidirectional search

The algorithms we have covered so far start at an initial state and can reach any one of multiple possible goal states. An alternative approach called **bidirectional search** simultaneously searches forward from the initial state and backwards from the goal state(s), hoping that the two searches will meet. The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d (e.g., 50,000 times less when $b = d = 10$).

Bidirectional search

For this to work, we need to keep track of two frontiers and two tables of reached states, and we need to be able to reason backwards: if state s' is a successor of s in the forward direction, then we need to know that s is a successor of s' in the backward direction. We have a solution when the two frontiers collide.[9]

9 In our implementation, the *reached* data structure supports a query asking whether a given state is a member, and the frontier data structure (a priority queue) does not, so we check for a collision using *reached*; but conceptually we are asking if the two frontiers have met up. The implementation can be extended to handle multiple goal states by loading the node for each goal state into the backwards frontier and backwards reached table.

There are many different versions of bidirectional search, just as there are many different unidirectional search algorithms. In this section, we describe bidirectional best-first search. Although there are two separate frontiers, the node to be expanded next is always one with a minimum value of the evaluation function, across either frontier. When the evaluation function is the path cost, we get bidirectional uniform-cost search, and if the cost of the optimal path is C^*, then no node with cost $> \frac{C^*}{2}$ will be expanded. This can result in a considerable speedup.

The general best-first bidirectional search algorithm is shown in Figure 3.14. We pass in two versions of the problem and the evaluation function, one in the forward direction (subscript F) and one in the backward direction (subscript B). When the evaluation function is the path cost, we know that the first solution found will be an optimal solution, but with different evaluation functions that is not necessarily true. Therefore, we keep track of the best solution found so far, and might have to update that several times before the TERMINATED test proves that there is no possible better solution remaining.

Figure 3.14

```
function BIBF-SEARCH(problem_F, f_F, problem_B, f_B) returns a solution node, or failure
  node_F ← NODE(problem_F.INITIAL)                    // Node for a start state
  node_B ← NODE(problem_B.INITIAL)                    // Node for a goal state
  frontier_F ← a priority queue ordered by f_F, with node_F as an element
  frontier_B ← a priority queue ordered by f_B, with node_B as an element
  reached_F ← a lookup table, with one key node_F.STATE and value node_F
  reached_B ← a lookup table, with one key node_B.STATE and value node_B
  solution ← failure
  while not TERMINATED(solution, frontier_F, frontier_B) do
    if f_F(TOP(frontier_F)) < f_B(TOP(frontier_B)) then
      solution ← PROCEED(F, problem_F frontier_F, reached_F, reached_B, solution)
    else solution ← PROCEED(B, problem_B, frontier_B, reached_B, reached_F, solution)
  return solution

function PROCEED(dir, problem, frontier, reached, reached_2, solution) returns a solution
        // Expand node on frontier; check against the other frontier in reached_2.
        // The variable "dir" is the direction: either F for forward or B for backward.
  node ← POP(frontier)
  for each child in EXPAND(problem, node) do
    s ← child.STATE
    if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
      reached[s] ← child
      add child to frontier
      if s is in reached_2 then
        solution_2 ← JOIN-NODES(dir, child, reached_2[s]))
        if PATH-COST(solution_2) < PATH-COST(solution) then
          solution ← solution_2
  return solution
```

Bidirectional best-first search keeps two frontiers and two tables of reached states. When a path in one frontier reaches a state that was also reached in the other half of the search, the two paths are joined (by the function JOIN-NODES) to form a solution. The first solution we get is not guaranteed to be the best; the function TERMINATED determines when to stop looking for new solutions.

3.4.6 Comparing uninformed search algorithms

Figure 3.15 compares uninformed search algorithms in terms of the four evaluation criteria set forth in Section 3.3.4. This comparison is for tree-like search versions which don't check for repeated states. For graph searches which do check, the main differences are that depth-first search is complete for finite state spaces, and the space and time complexities are bounded by the size of the state space (the number of vertices and edges, $|V| + |E|$).

Figure 3.15

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

Evaluation of search algorithms. b is the branching factor; m is the maximum depth of the search tree; d is the depth of the shallowest solution, or is m when there is no solution; ℓ is the depth limit. Superscript caveats are as follows: [1] complete if b is finite, and the state space either has a solution or is finite. [2] complete if all action costs are $\geq \varepsilon > 0$; [3] cost-optimal if action costs are all identical; [4] if both directions are breadth-first or uniform-cost.

3.5 Informed (Heuristic) Search Strategies

This section shows how an **informed search** strategy—one that uses domain-specific hints about the location of goals—can find solutions more efficiently than an uninformed strategy. The hints come in the form of a **heuristic function**, denoted $h(n)$:[10]

10 It may seem odd that the heuristic function operates on a node, when all it really needs is the node's state. It is traditional to use $h(n)$ rather than $h(s)$ to be consistent with the evaluation function $f(n)$ and the path cost $g(n)$.

$$h(n) = \text{ estimated cost of the cheapest path from the state at node } n \text{ to a goal state.}$$

Informed search

Heuristic function

For example, in route-finding problems, we can estimate the distance from the current state to a goal by computing the straight-line distance on the map between the two points. We study heuristics and where they come from in more detail in Section 3.6.

3.5.1 Greedy best-first search

Greedy best-first search is a form of best-first search that expands first the node with the lowest $h(n)$ value—the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly. So the evaluation function $f(n) = h(n)$.

Greedy best-first search

Let us see how this works for route-finding problems in Romania; we use the **straight-line distance** heuristic, which we will call h_{SLD}. If the goal is Bucharest, we need to know the straight-line distances to Bucharest, which are shown in Figure 3.16. For example, $h_{SLD}(Arad) = 366$. Notice that the values of h_{SLD} cannot be computed from the problem description itself (that is, the ACTIONS and RESULT functions). Moreover, it takes a certain amount of world knowledge to know that h_{SLD} is correlated with actual road distances and is, therefore, a useful heuristic.

Figure 3.16

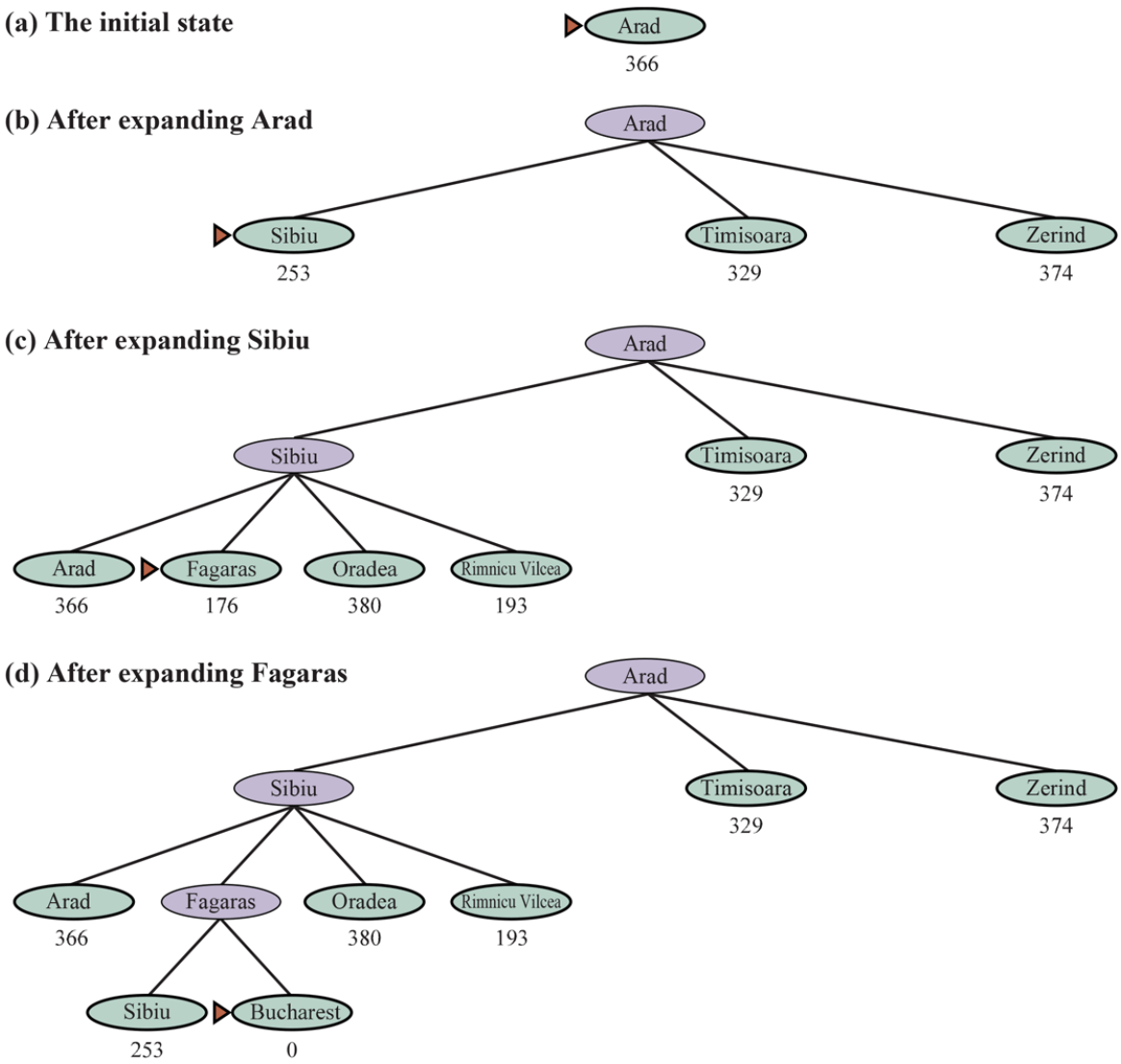
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Values of h_{SLD}—straight-line distances to Bucharest.

Straight-line distance

Figure 3.17 shows the progress of a greedy best-first search using h_{SLD} to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu because the heuristic says it is closer to Bucharest than is either Zerind or Timisoara. The next node to be expanded will be Fagaras because it is now closest according to the heuristic. Fagaras in turn generates Bucharest, which is the goal. For this particular problem, greedy best-first search using h_{SLD} finds a solution without ever expanding a node that is not on the solution path. The solution it found does not have optimal cost, however: the path via Sibiu and Fagaras to Bucharest is 32 miles longer than the path through Rimnicu Vilcea and Pitesti. This is why the algorithm is called "greedy"—on each iteration it tries to get as close to a goal as it can, but greediness can lead to worse results than being careful.

Figure 3.17



Stages in a greedy best-first tree-like search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Greedy best-first graph search is complete in finite state spaces, but not in infinite ones. The worst-case time and space complexity is $O(|V|)$. With a good heuristic function, however, the complexity can be reduced substantially, on certain problems reaching $O(bm)$.

3.5.2 A* search

The most common informed search algorithm is **A* search** (pronounced "A-star search"), a best-first search that uses the evaluation function

$$f(n) = g(n) + h(n)$$

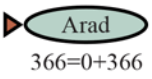
where $g(n)$ is the path cost from the initial state to node n, and $h(n)$ is the *estimated* cost of the shortest path from n to a goal state, so we have

$$f(n) = \text{ estimated cost of the best path that continues from } n \text{ to a goal.}$$

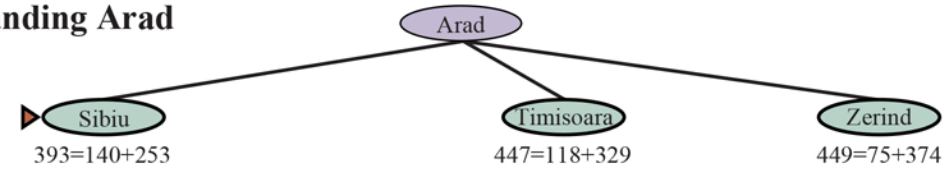
In Figure 3.18, we show the progress of an A* search with the goal of reaching Bucharest. The values of g are computed from the action costs in Figure 3.1, and the values of h_{SLD} are given in Figure 3.16. Notice that Bucharest first appears on the frontier at step (e), but it is not selected for expansion (and thus not detected as a solution) because at $f = 450$ it is not the lowest-cost node on the frontier—that would be Pitesti, at $f = 417$. Another way to say this is that there *might* be a solution through Pitesti whose cost is as low as 417, so the algorithm will not settle for a solution that costs 450. At step (f), a different path to Bucharest is now the lowest-cost node, at $f = 418$, so it is selected and detected as the optimal solution.

Figure 3.18

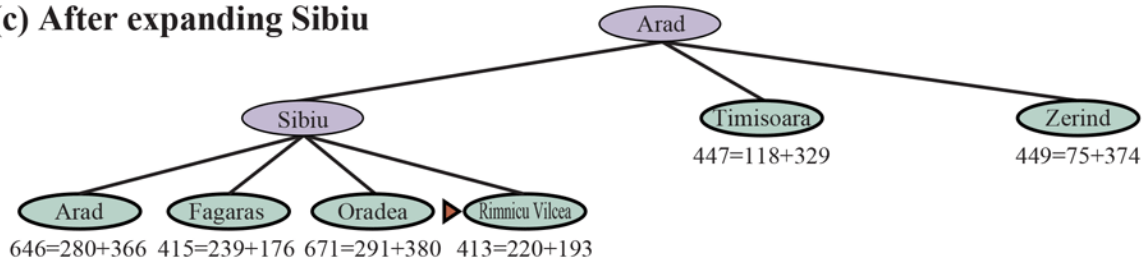
(a) The initial state



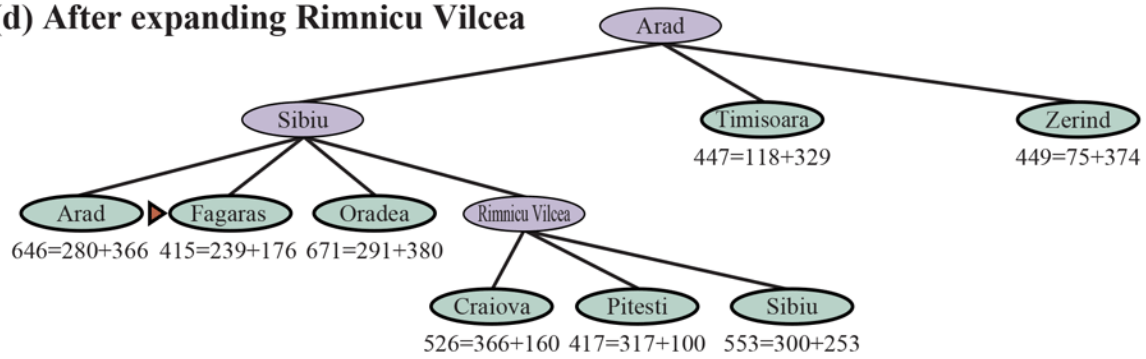
(b) After expanding Arad



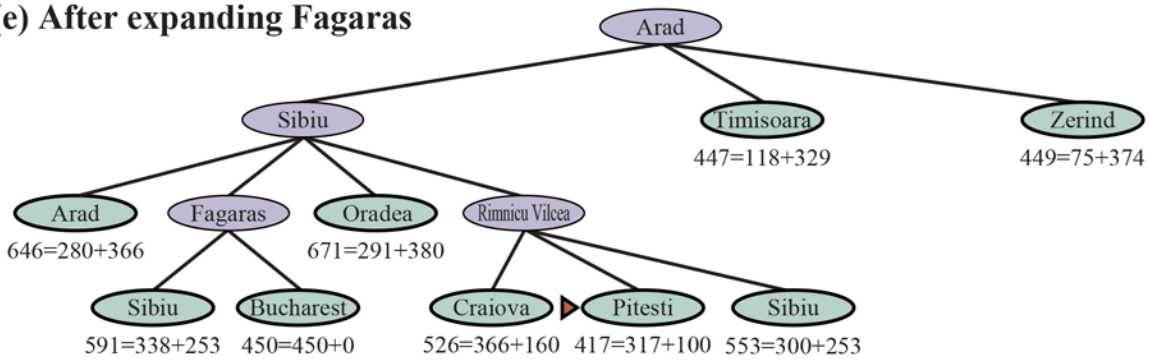
(c) After expanding Sibiu



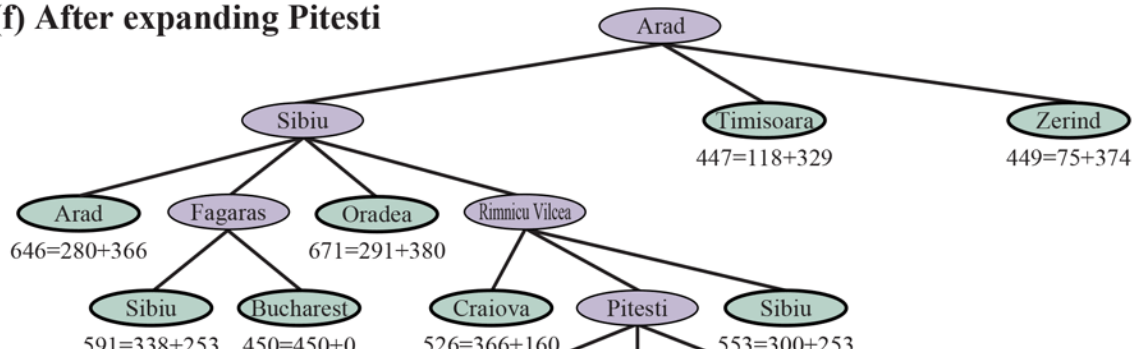
(d) After expanding Rimnicu Vilcea

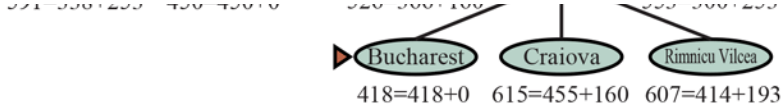


(e) After expanding Fagaras



(f) After expanding Pitesti





Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.16.

Admissible heuristic

A* search is complete.[11] Whether A* is cost-optimal depends on certain properties of the heuristic. A key property is **admissibility**: an **admissible heuristic** is one that *never overestimates* the cost to reach a goal. (An admissible heuristic is therefore *optimistic*.) With an admissible heuristic, A* is cost-optimal, which we can show with a proof by contradiction. Suppose the optimal path has cost C^*, but the algorithm returns a path with cost $C > C^*$. Then there must be some node n which is on the optimal path and is unexpanded (because if all the nodes on the optimal path had been expanded, then we would have returned that optimal solution). So then, using the notation $g^*(n)$ to mean the cost of the optimal path from the start to n, and $h^*(n)$ to mean the cost of the optimal path from n to the nearest goal, we have:

11 Again, assuming all action costs are $>\epsilon>0$, and the state space either has a solution or is finite.

$$\begin{aligned}
f(n) &> C^* \quad (\text{otherwise } n \text{ would have been expanded}) \\
f(n) &= g(n) + h(n) \quad (\text{by definition}) \\
f(n) &= g^*(n) + h(n) \quad (\text{because } n \text{ is on an optimal path}) \\
f(n) &\leq g^*(n) + h^*(n) \quad (\text{because of admissibility}, h(n) \leq h^*(n)) \\
f(n) &\leq C^* \quad (\text{by definition}, C^* = g^*(n) + h^*(n))
\end{aligned}$$

The first and last lines form a contradiction, so the supposition that the algorithm could return a suboptimal path must be wrong—it must be that A* returns only cost-optimal paths.

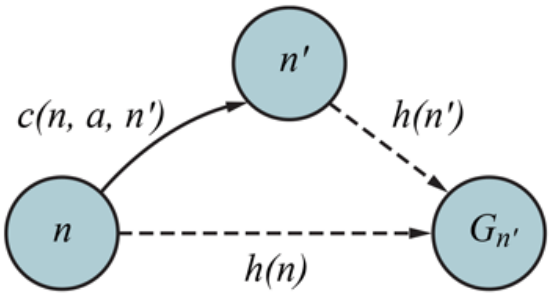
A slightly stronger property is called **consistency**. A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by an action a, we have:

$$h(n) \leq c(n, a, n') + h(n').$$

Consistency

This is a form of the **triangle inequality**, which stipulates that a side of a triangle cannot be longer than the sum of the other two sides (see Figure 3.19). An example of a consistent heuristic is the straight-line distance h_{SLD} that we used in getting to Bucharest.

Figure 3.19



Triangle inequality: If the heuristic h is **consistent**, then the single number $h(n)$ will be less than the sum of the cost $c(n, a, a')$ of the action from n to n' plus the heuristic estimate $h(n')$.

Triangle inequality

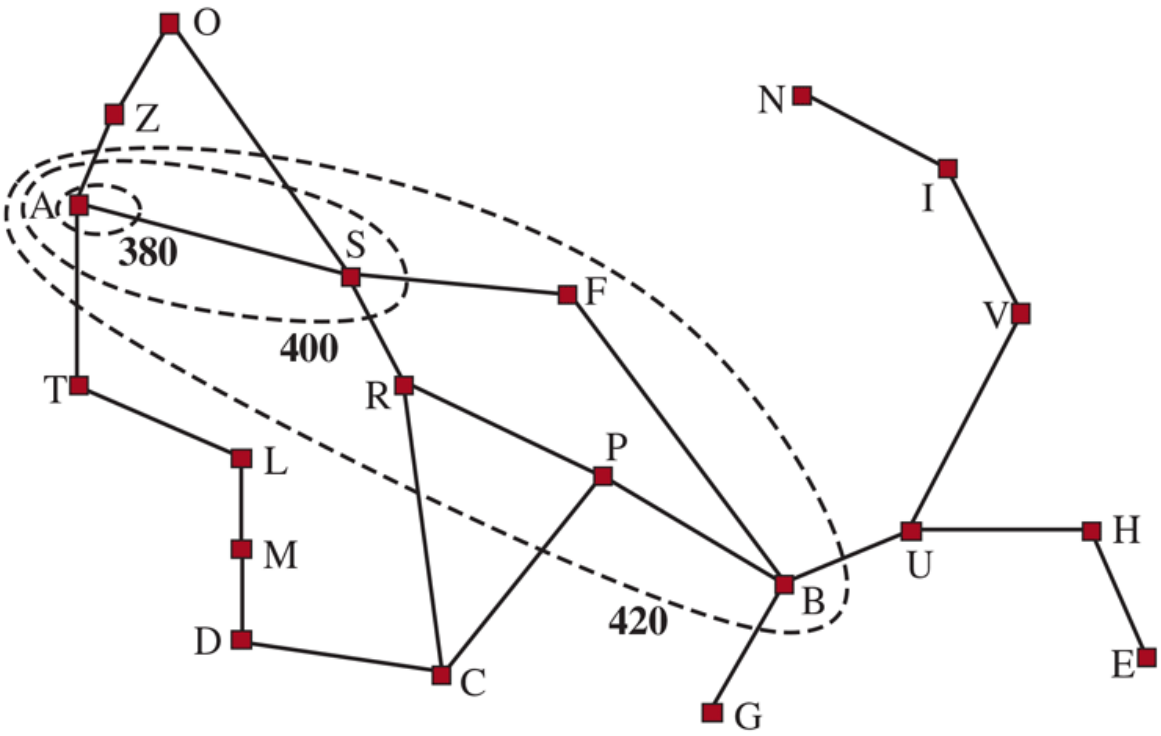
Every consistent heuristic is admissible (but not vice versa), so with a consistent heuristic, A* is cost-optimal. In addition, with a consistent heuristic, the first time we reach a state it will be on an optimal path, so we never have to re-add a state to the frontier, and never have to change an entry in *reached*. But with an inconsistent heuristic, we may end up with multiple paths reaching the same state, and if each new path has a lower path cost than the previous one, then we will end up with multiple nodes for that state in the frontier, costing us both time and space. Because of that, some implementations of A* take care to only enter a state into the frontier once, and if a better path to the state is found, all the successors of the state are updated (which requires that nodes have child pointers as well as parent pointers). These complications have led many implementers to avoid inconsistent heuristics, but Felner *et al.* (2011) argues that the worst effects rarely happen in practice, and one shouldn't be afraid of inconsistent heuristics.

With an inadmissible heuristic, A* may or may not be cost-optimal. Here are two cases where it is: First, if there is even one cost-optimal path on which $h(n)$ is admissible for all nodes n on the path, then that path will be found, no matter what the heuristic says for states off the path. Second, if the optimal solution has cost C^*, and the second-best has cost C_2, and if $h(n)$ overestimates some costs, but never by more than $C_2 - C^*$, then A* is guaranteed to return cost-optimal solutions.

3.5.3 Search contours

A useful way to visualize a search is to draw **contours** in the state space, just like the contours in a topographic map. Figure 3.20 shows an example. Inside the contour labeled 400, all nodes have $f(n) = g(n) + h(n) \leq 400$, and so on. Then, because A* expands the frontier node of lowest f-cost, we can see that an A* search fans out from the start node, adding nodes in concentric bands of increasing f-cost.

Figure 3.20



Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have $f = g + h$ costs less than or equal to the contour value.

Contour

With uniform-cost search, we also have contours, but of g-cost, not $g + h$. The contours with uniform-cost search will be "circular" around the start state, spreading out equally in all directions with no preference towards the goal. With A* search using a good heuristic, the $g + h$ bands will stretch toward a goal state (as in Figure 3.20) and become more narrowly focused around an optimal path.

It should be clear that as you extend a path, the g costs are **monotonic**: the path cost always increases as you go along a path, because action costs are always positive.[12] Therefore you get concentric contour lines that don't cross each other, and if you choose to draw the lines fine enough, you can put a line between any two nodes on any path.

12 Technically, we say "strictly monotonic" for costs that always increase, and "monotonic" for costs that never decrease, but might remain the same.

Monotonic

But it is not obvious whether the $f = g + h$ cost will monotonically increase. As you extend a path from n to n', the cost goes from $g(n) + h(n)$ to $g(n) + c(n, a, n') + h(n')$. Canceling out the $g(n)$ term, we see that the path's cost will be monotonically increasing if and only if $h(n) \leq c(n, a, n') + h(n')$; in other words if and only if the heuristic is consistent.[13] But note that a path might contribute several nodes in a row with the same $g(n) + h(n)$ score; this will happen whenever the decrease in h is exactly equal to the action cost just taken (for example, in a grid problem, when n is in the same row as the goal and you take a step towards the goal, g is increased by 1 and h is decreased by 1). If C^* is the cost of the optimal solution path, then we can say the following:

13 In fact, the term "monotonic heuristic" is a synonym for "consistent heuristic." The two ideas were developed independently, and then it was proved that they are equivalent (Pearl, 1984).

- A* expands all nodes that can be reached from the initial state on a path where every node on the path has $f(n) < C^*$. We say these are **surely expanded nodes**.

Surely expanded nodes

- A* might then expand some of the nodes right on the "goal contour" (where $f(n) = C^*$) before selecting a goal node.
- A* expands no nodes with $f(n) > C^*$.

We say that A* with a consistent heuristic is **optimally efficient** in the sense that any algorithm that extends search paths from the initial state, and uses the same heuristic information, must expand all nodes that are surely expanded by A* (because any one of them could have been part of an optimal solution). Among the nodes with $f(n) = C^*$, one algorithm could get lucky and choose the optimal one first while another algorithm is unlucky; we don't consider this difference in defining optimal efficiency.

Optimally efficient

A* is efficient because it **prunes** away search tree nodes that are not necessary for finding an optimal solution. In Figure 3.18(b) we see that Timisoara has $f = 447$ and Zerind has $f = 449$. Even though they are children of the root and would be among the first nodes expanded by uniform-cost or breadth-first search, they are never expanded by A* search because the solution with $f = 418$ is found first. The concept of pruning—eliminating possibilities from consideration without having to examine them—is important for many areas of AI.

Pruning

That A* search is complete, cost-optimal, and optimally efficient among all such algorithms is rather satisfying. Unfortunately, it does not mean that A* is the answer to all our searching needs. The catch is that for many problems, the number of nodes expanded can be exponential in the length of the solution. For example, consider a version of the vacuum world with a super-powerful vacuum that can clean up any one square at a cost of 1 unit, without even having to visit the square; in that scenario, squares can be cleaned in any order. With N initially dirty squares, there are 2^N states where some subset has been cleaned; all of those states are on an optimal solution path, and hence satisfy $f(n) < C^*$, so all of them would be visited by A*.

3.5.4 Satisficing search: Inadmissible heuristics and weighted A*

Inadmissible heuristic

A* search has many good qualities, but it expands a lot of nodes. We can explore fewer nodes (taking less time and space) if we are willing to accept solutions that are suboptimal, but are "good enough"—what we call **satisficing** solutions. If we allow A* search to use an **inadmissible heuristic**—one that may overestimate—then we risk missing the optimal solution, but the heuristic can potentially be more accurate, thereby reducing the number of nodes expanded. For example, road engineers know the concept of a **detour index**, which is a multiplier applied to the straight-line distance to account for the typical curvature of roads. A detour index of 1.3 means that if two cities are 10 miles apart in straight-line distance, a good estimate of the best path between them is 13 miles. For most localities, the detour index ranges between 1.2 and 1.6.

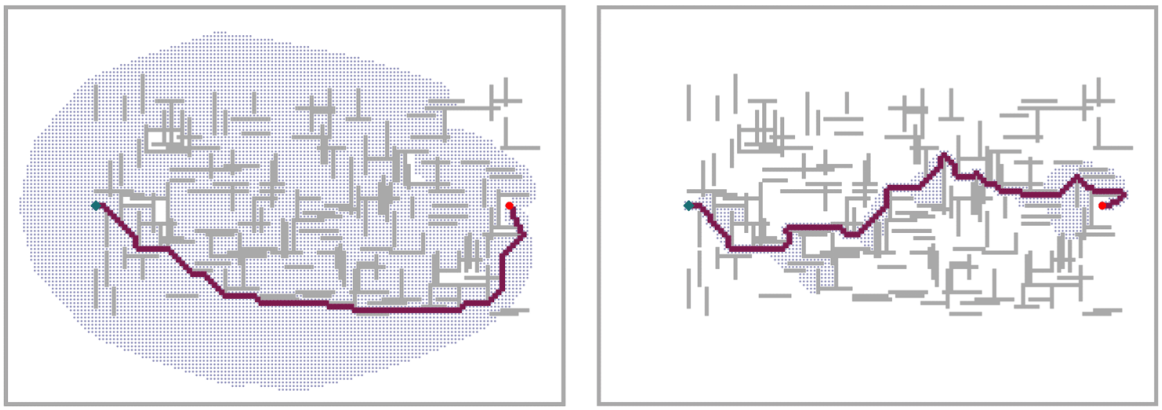
Detour index

We can apply this idea to any problem, not just ones involving roads, with an approach called **weighted A* search** where we weight the heuristic value more heavily, giving us the evaluation function $f(n) = g(n) + W \times h(n)$, for some $W > 1$.

Weighted A search*

Figure 3.21 shows a search problem on a grid world. In (a), an A* search finds the optimal solution, but has to explore a large portion of the state space to find it. In (b), a weighted A* search finds a solution that is slightly costlier, but the search time is much faster. We see that the weighted search focuses the contour of reached states towards a goal. That means that fewer states are explored, but if the optimal path ever strays outside of the weighted search's contour (as it does in this case), then the optimal path will not be found. In general, if the optimal solution costs C^*, a weighted A* search will find a solution that costs somewhere between C^* and $W \times C^*$; but in practice we usually get results much closer to C^* than $W \times C^*$.

Figure 3.21



(a) (b)

Two searches on the same grid: (a) an A* search and (b) a weighted A* search with weight $W = 2$. The gray bars are obstacles, the purple line is the path from the green start to red goal, and the small dots are states that were reached by each search. On this particular problem, weighted A* explores 7 times fewer states and finds a path that is 5% more costly.

We have considered searches that evaluate states by combining g and h in various ways; weighted A* can be seen as a generalization of the others:

$$\begin{aligned}
\text{A* search:} \quad & g(n) + h(n) \quad && (W = 1) \\
\text{Uniform-cost search:} \quad & g(n) && (W = 0) \\
\text{Greedy best-first search:} \quad & h(n) && (W = \infty) \\
\text{Weighted A* search:} \quad & g(n) + W \times h(n) && (1 < W < \infty)
\end{aligned}$$

You could call weighted A* "somewhat-greedy search": like greedy best-first search, it focuses the search towards a goal; on the other hand, it won't ignore the path cost completely, and will suspend a path that is making little progress at great cost.

There are a variety of suboptimal search algorithms, which can be characterized by the criteria for what counts as "good enough." In **bounded suboptimal search**, we look for a solution that is guaranteed to be within a constant factor W of the optimal cost. Weighted A* provides this guarantee. In **bounded-cost search**, we look for a solution whose cost is less than some constant C. And in **unbounded-cost search**, we accept a solution of any cost, as long as we can find it quickly.

Bounded suboptimal search

Bounded-cost search

Unbounded-cost search

An example of an unbounded-cost search algorithm is **speedy search**, which is a version of greedy best-first search that uses as a heuristic the estimated number of actions required to reach a goal, regardless of the cost of those actions. Thus, for problems where all actions

have the same cost it is the same as greedy best-first search, but when actions have different costs, it tends to lead the search to find a solution quickly, even if it might have a high cost.

Speedy search

3.5.5 Memory-bounded search

The main issue with A* is its use of memory. In this section we'll cover some implementation tricks that save space, and then some entirely new algorithms that take better advantage of the available space.

Memory is split between the *frontier* and the *reached* states. In our implementation of best-first search, a state that is on the frontier is stored in two places: as a node in the frontier (so we can decide what to expand next) and as an entry in the table of reached states (so we know if we have visited the state before). For many problems (such as exploring a grid), this duplication is not a concern, because the size of *frontier* is much smaller than *reached*, so duplicating the states in the frontier requires a comparatively trivial amount of memory. But some implementations keep a state in only one of the two places, saving a bit of space at the cost of complicating (and perhaps slowing down) the algorithm.

Another possibility is to remove states from *reached* when we can prove that they are no longer needed. For some problems, we can use the separation property (Figure 3.6 on page 72), along with the prohibition of U-turn actions, to ensure that all actions either move outwards from the frontier or onto another frontier state. In that case, we need only check the frontier for redundant paths, and we can eliminate the *reached* table.

For other problems, we can keep **reference counts** of the number of times a state has been reached, and remove it from the *reached* table when there are no more ways to reach the state. For example, on a grid world where each state can be reached only from its four neighbors, once we have reached a state four times, we can remove it from the table.

Reference count

Now let's consider new algorithms that are designed to conserve memory usage.

Beam search limits the size of the frontier. The easiest approach is to keep only the k nodes with the best f-scores, discarding any other expanded nodes. This of course makes the search incomplete and suboptimal, but we can choose k to make good use of available memory, and the algorithm executes fast because it expands fewer nodes. For many problems it can find good near-optimal solutions. You can think of uniform-cost or A* search as spreading out everywhere in concentric contours, and think of beam search as exploring only a focused portion of those contours, the portion that contains the k best candidates.

Beam search

An alternative version of beam search doesn't keep a strict limit on the size of the frontier but instead keeps every node whose f-score is within δ of the best f-score. That way, when there are a few strong-scoring nodes only a few will be kept, but if there are no strong nodes then more will be kept until a strong one emerges.

Iterative-deepening A* search (IDA*) is to A* what iterative-deepening search is to depth-first: IDA* gives us the benefits of A* without the requirement to keep all reached states in memory, at a cost of visiting some states multiple times. It is a very important and commonly used algorithm for problems that do not fit in memory.

Iterative-deepening A search*

In standard iterative deepening the cutoff is the depth, which is increased by one each iteration. In IDA* the cutoff is the f-cost ($g + h$); at each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration. In other words, each iteration exhaustively searches an f-contour, finds a node just beyond that contour, and uses that node's f-cost as the next contour. For problems like the 8-puzzle where each path's f-cost is an integer, this works very well, resulting in steady progress towards the goal each iteration. If the optimal solution has cost C^*, then there can be no more than C^* iterations (for example, no more than 31 iterations on the hardest 8-puzzle problems). But for a problem where every node has a different f-cost, each new contour might contain only one new node, and the number of iterations could be equal to the number of states.

Recursive best-first search (RBFS) (Figure 3.22) attempts to mimic the operation of standard best-first search, but using only linear space. RBFS resembles a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the f_limit variable to keep track of the f-value of the best *alternative* path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with a **backed-up value**—the best f-value of its children. In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it's worth reexpanding the subtree at some later time. Figure 3.23 shows how RBFS reaches Bucharest.

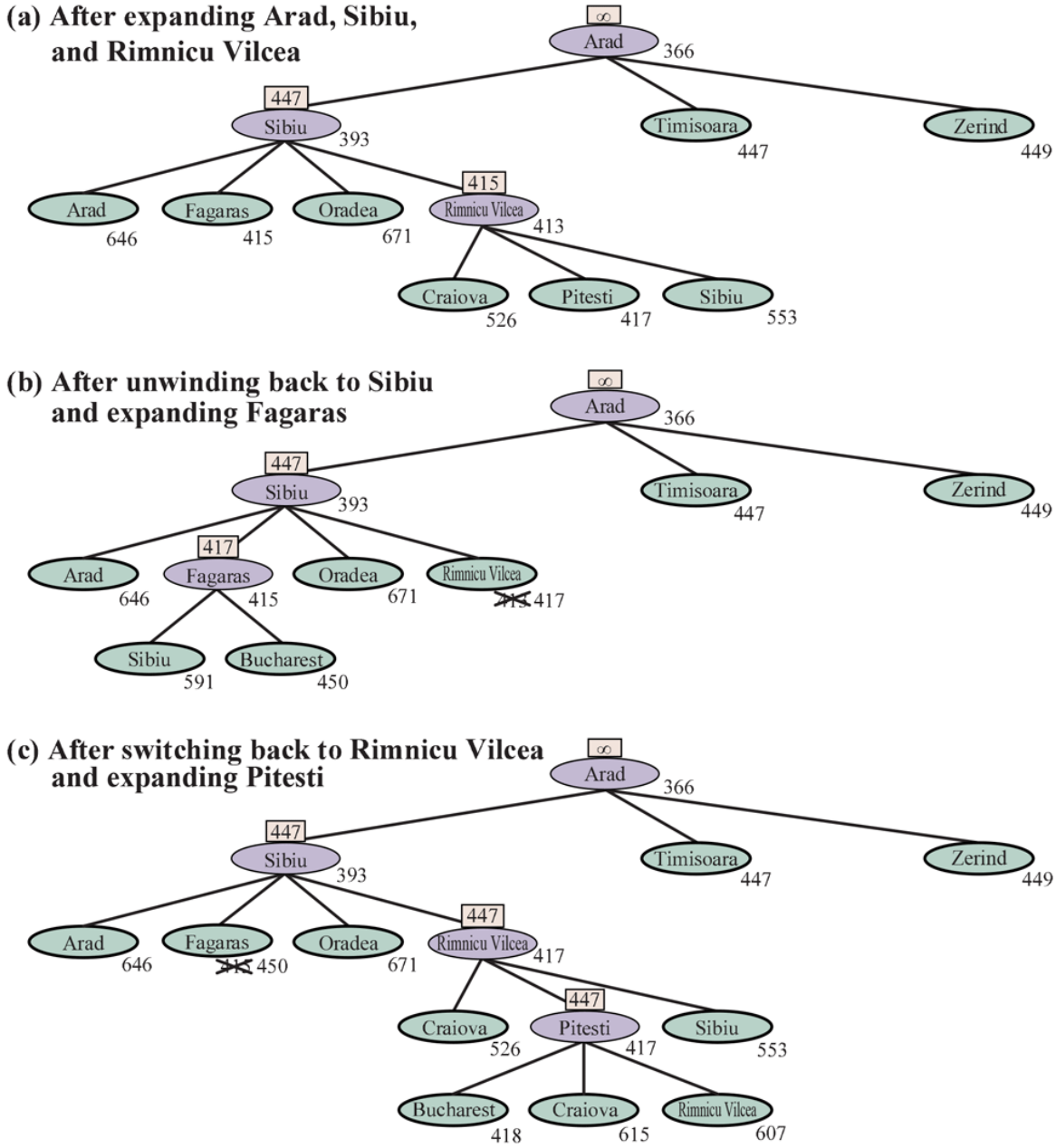
Figure 3.22

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution or failure
    solution, fvalue ← RBFS(problem, NODE(problem.INITIAL), ∞)
  return solution

function RBFS(problem, node, f_limit) returns a solution or failure, and a new f-cost limit
  if problem.IS-GOAL(node.STATE) then return node
  successors ← LIST(EXPAND(node))
  if successors is empty then return failure, ∞
  for each s in successors do        // update f with value from previous search
      s.f ← max(s.PATH-COST + h(s), node.f))
  while true do
      best ← the node in successors with lowest f-value
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result, best.f
```

The algorithm for recursive best-first search.

Figure 3.23



Stages in an RBFS search for the shortest route to Bucharest. The *f-limit* value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

Recursive best-first search

Backed-up value

RBFS is somewhat more efficient than IDA*, but still suffers from excessive node regeneration. In the example in Figure 3.23, RBFS follows the path via Rimnicu Vilcea, then "changes its mind" and tries Fagaras, and then changes its mind back again. These mind changes occur because every time the current best path is extended, its f-value is likely to increase—h is usually less optimistic for nodes closer to a goal. When this happens, the second-best path might become the best path, so the search has to backtrack to follow it. Each mind change corresponds to an iteration of IDA* and could require many reexpansions of forgotten nodes to recreate the best path and extend it one more node.

RBFS is optimal if the heuristic function $h(n)$ is admissible. Its space complexity is linear in the depth of the deepest optimal solution, but its time complexity is rather difficult to characterize: it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded. It expands nodes in order of increasing f-score, even if f is nonmonotonic.

IDA* and RBFS suffer from using *too little* memory. Between iterations, IDA* retains only a single number: the current f-cost limit. RBFS retains more information in memory, but it uses only linear space: even if more memory were available, RBFS has no way to make use of it. Because they forget most of what they have done, both algorithms may end up reexploring the same states many times over.

It seems sensible, therefore, to determine how much memory we have available, and allow an algorithm to use all of it. Two algorithms that do this are **MA*** (memory-bounded A*) and **SMA*** (simplified MA*). SMA* is—well—simpler, so we will describe it. SMA* proceeds just like A*, expanding the best leaf until memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA* always drops the *worst* leaf node—the one with the highest f-value. Like RBFS, SMA* then backs up the value of the forgotten node to its parent. In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree. With this information, SMA* regenerates the subtree only when all other paths have been shown to look worse than the path it has forgotten. Another way of saying this is that if all the descendants of a node n are forgotten, then we

will not know which way to go from n, but we will still have an idea of how worthwhile it is to go anywhere from n.

---

*MA**

---

*SMA**

The complete algorithm is described in the online code repository accompanying this book. There is one subtlety worth mentioning. We said that SMA* expands the best leaf and deletes the worst leaf. What if *all* the leaf nodes have the same f-value? To avoid selecting the same node for deletion and expansion, SMA* expands the *newest* best leaf and deletes the *oldest* worst leaf. These coincide when there is only one leaf, but in that case, the current search tree must be a single path from root to leaf that fills all of memory. If the leaf is not a goal node, then *even if it is on an optimal solution path*, that solution is not reachable with the available memory. Therefore, the node can be discarded exactly as if it had no successors.

SMA* is complete if there is any reachable solution—that is, if d, the depth of the shallowest goal node, is less than the memory size (expressed in nodes). It is optimal if any optimal solution is reachable; otherwise, it returns the best reachable solution. In practical terms, SMA* is a fairly robust choice for finding optimal solutions, particularly when the state space is a graph, action costs are not uniform, and node generation is expensive compared to the overhead of maintaining the frontier and the reached set.

On very hard problems, however, it will often be the case that SMA* is forced to switch back and forth continually among many candidate solution paths, only a small subset of which can fit in memory. (This resembles the problem of **thrashing** in disk paging systems.) Then the extra time required for repeated regeneration of the same nodes means that problems that would be practically solvable by A*, given unlimited memory, become intractable for SMA*. That is to say, *memory limitations can make a problem intractable from the point of view of computation time*. Although no current theory explains the tradeoff

between time and memory, it seems that this is an inescapable problem. The only way out is to drop the optimality requirement.

Thrashing

3.5.6 Bidirectional heuristic search

With unidirectional best-first search, we saw that using $f(n) = g(n) + h(n)$ as the evaluation function gives us an A* search that is guaranteed to find optimal-cost solutions (assuming an admissible h) while being optimally efficient in the number of nodes expanded.

With bidirectional best-first search we could also try using $f(n) = g(n) + h(n)$, but unfortunately there is no guarantee that this would lead to an optimal-cost solution, nor that it would be optimally efficient, even with an admissible heuristic. With bidirectional search, it turns out that it is not individual nodes but rather *pairs* of nodes (one from each frontier) that can be proved to be surely expanded, so any proof of efficiency will have to consider pairs of nodes (Eckerle *et al.*, 2017).

We'll start with some new notation. We use $f_F(n) = g_F(n) + h_F(n)$ for nodes going in the forward direction (with the initial state as root) and $f_B(n) = g_B(n) + h_B(n)$ for nodes in the backward direction (with a goal state as root). Although both forward and backward searches are solving the same problem, they have different evaluation functions because, for example, the heuristics are different depending on whether you are striving for the goal or for the initial state. We'll assume admissible heuristics.

Consider a forward path from the initial state to a node m and a backward path from the goal to a node n. We can define a lower bound on the cost of a solution that follows the path from the initial state to m, then somehow gets to n, then follows the path to the goal as

$$lb(m, n) = \max(g_F\ (m) + g_B(n),\ f_F(m),\ f_B(n))$$

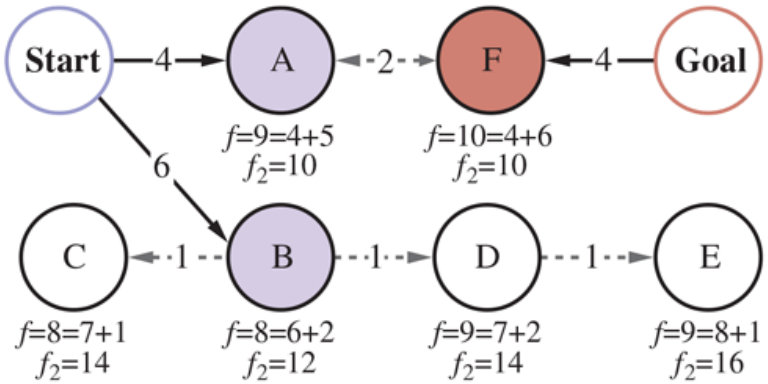
In other words, the cost of such a path must be at least as large as the sum of the path costs of the two parts (because the remaining connection between them must have nonnegative

cost), and the cost must also be at least as much as the estimated f cost of either part (because the heuristic estimates are optimistic). Given that, the theorem is that for any pair of nodes m, n with $lb(m, n)$ less than the optimal cost C^*, we must expand either m or n, because the path that goes through both of them is a potential optimal solution. The difficulty is that we don't know for sure which node is best to expand, and therefore no bidirectional search algorithm can be guaranteed to be optimally efficient—any algorithm might expand up to twice the minimum number of nodes if it always chooses the wrong member of a pair to expand first. Some bidirectional heuristic search algorithms explicitly manage a queue of (m, n) pairs, but we will stick with bidirectional best-first search (Figure 3.14), which has two frontier priority queues, and give it an evaluation function that mimics the lb criteria:

$$f_2(n) = \max(2g(n), g(n) + h(n))$$

The node to expand next will be the one that minimizes this f_2 value; the node can come from either frontier. This f_2 function guarantees that we will never expand a node (from either frontier) with $g(n) > \frac{C^*}{2}$. We say the two halves of the search "meet in the middle" in the sense that when the two frontiers touch, no node inside of either frontier has a path cost greater than the bound $\frac{C^*}{2}$. Figure 3.24 works through an example bidirectional search.

Figure 3.24



Bidirectional search maintains two frontiers: on the left, nodes A and B are successors of Start; on the right, node F is an inverse successor of Goal. Each node is labeled with $f = g + h$ values and the $f_2 = \max(2g, g + h)$ value. (The g values are the sum of the action costs as shown on each arrow; the h values are arbitrary and cannot be derived from anything in the figure.) The optimal solution, Start-A-F-Goal, has cost $C^* = 4 + 2 + 4 = 10$, so that means that a meet-in-the-middle bidirectional algorithm should not expand any node with $g > \frac{C^*}{2} = 5$; and indeed the next node to be expanded would be A or F (each with $g = 4$), leading us to an optimal solution. If we expanded the node with lowest f cost first, then B and C would come next, and D and E would be tied with A, but they all have $g > \frac{C^*}{2}$ and thus are never expanded when f_2 is the evaluation function.

Front-to-end

Front-to-front

We have described an approach where the h_F heuristic estimates the distance to the goal (or, when the problem has multiple goal states, the distance to the closest goal) and h_B estimates the distance to the start. This is called a **front-to-end** search. An alternative, called **front-to-front** search, attempts to estimate the distance to the other frontier. Clearly, if a frontier has millions of nodes, it would be inefficient to apply the heuristic function to every one of them and take the minimum. But it can work to sample a few nodes from the frontier. In certain specific problem domains it is possible to *summarize* the frontier—for example, in a grid search problem, we can incrementally compute a bounding box of the frontier, and use as a heuristic the distance to the bounding box.

Bidirectional search is sometimes more efficient than unidirectional search, sometimes not. In general, if we have a very good heuristic, then A* search produces search contours that are focused on the goal, and adding bidirectional search does not help much. With an average heuristic, bidirectional search that meets in the middle tends to expand fewer nodes and is preferred. In the worst case of a poor heuristic, the search is no longer focused on the goal, and bidirectional search has the same asymptotic complexity as A*. Bidirectional search with the f_2 evaluation function and an admissible heuristic h is complete and optimal.